Need introduction here

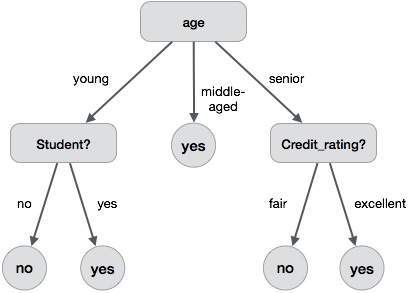
Solving a problem such as the one proposed on Kaggle generally requires many steps, as data usually does not come in the desired format and involves substantial challenges. At first, we had to set up the system with appropriate libraries, then define the problem, transform the data to make it digestible for the algorithm, perform the algorithm and analyse the resulting outcome, tune it, and finally present some potential improvements.

# Decision tree Engine

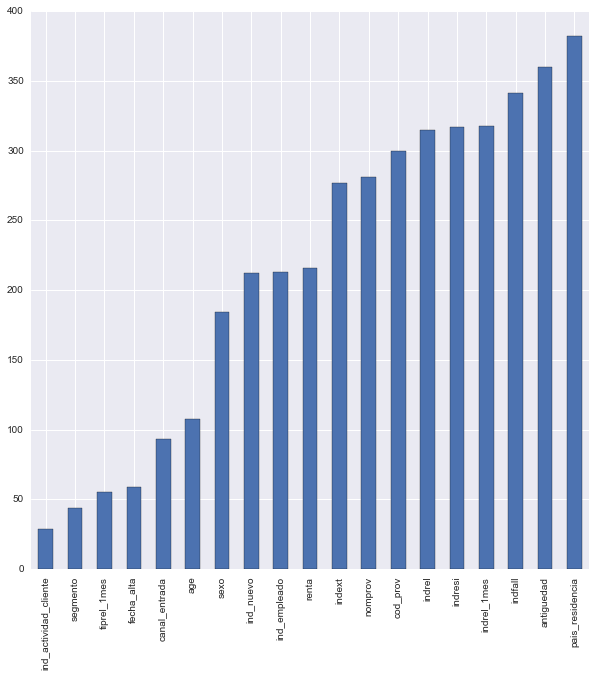
## 1.1 Setting up the system

After initial planning and theory research behind the workings of decision tree engines, it was deemed necessary to stage the data and explore its content, specifically to find the most important and relevant features to implement.

These features are shown as the vertices in Figure (1), and are the characteristics of the customer that the model uses to recommend the products.

**Fig(1) A visual representation of a simpledecision tree similar to the ones built by our model to predict products.**

This involved data engineering and cleansing, specifically imputing along numerous columns in the data set, followed by the use of **sklean.selectKbest** library to determine the most influential data features on what the customer is likely to buy.

**SelectKbest** library involves looking at the target variables present, and the correlation between those and the features provided, giving a score of the most influential ones. The results of this analysis are shown in figure 1.

**Fig (1): a graph representing the importance of the features on the total number of products bought by customers, in a negative scale, meaning the lowest valued features are the most influential ones.**

With the most important features discovered, work underwent on producing a simple working decision tree model, with the plan of using the predicted probabilities for products as a measure for our recommender engine.

## 1.2 Results

The engine managed to produce relatively accurate prediction based on the training set provided (~85% accuracy), returning the probabilities of predictions and formatting them into an accepted format was deemed to be too inefficient for the task. Once the code was completed (alterations meant automatic cleansing and engineering of both training and testing data sets), work shifted towards a more suitable engine.

# XGBoost

Notice that this recommendation engine is based around a publicly available script provided on Kaggle by author SRK.

## 2.1 Defining the problem

Building a recommendation engine can be achieved in different manners. It can rely on the properties of the items (content-based filtering) or on the similarity of products between users (collaborative filtering). We used a hybrid model to achieve maximal results, blending both the characteristics of the customers and the similarity of the product they purchased.

## 2.2 Setting up the system

Initial work involved importing the script and relevant programs needed to run the recommender engine (XGBoost is a machine-learning algorithm, which is notoriously difficult to install on windows).

## 2.3 Data cleaning

XGBoost manages only numerical vectors. As most of our data were categorical variables, we transformed them in an ingestible input for the algorithm via the one-hot encoding method.

The numbers of records processed through the ML algorithm were minimized by only looking at the customers that have purchased products within our specified date range, rather than all customers. Since the purpose of this project is to predict products customers are most likely to buy, rather than likelihood of them owning a product, it makes sense to only process products purchased.

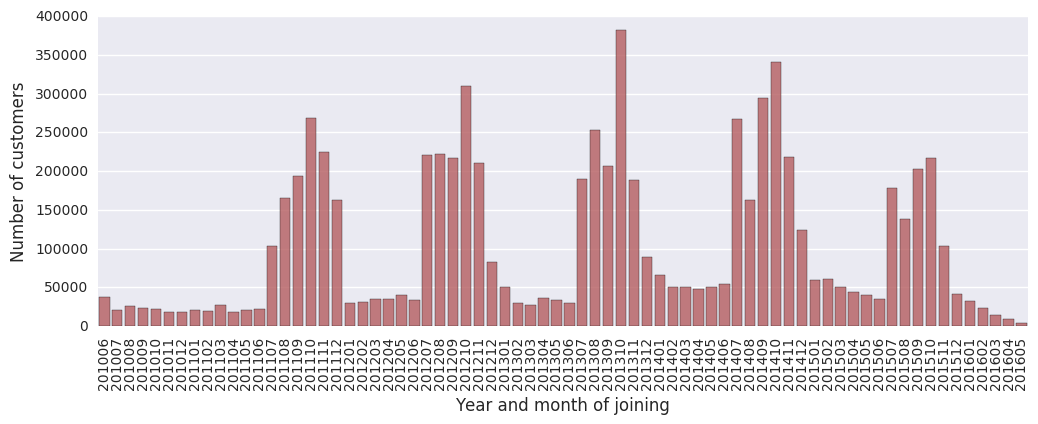
## 2.4 Feature Engineering

There seems to be two important determinants in the performance achieved in a machine learning solution: features engineering and choice of algorithm.

A blend of the customers defining features, such as Age, Activity and Region were used as the typical features. The customer’s product list from the same time the previous year were also listed, in order to capture seasonality.

As well as product seasonality (which for June in Spain seemed to be an influential factor since this is the month tax returns are due) we also wanted to portray trend effects, specifically how the customers buying patterns fluctuate leading up the to the months we want to predict.

This was achieved by adding in lag\_product features, comprising of a list of product ownership per customer dating back the previous months.



**Fig (2) Diagram showing the total number of customers, split by month and year. As clearly visible, there is strong seasonality with number of customers spiking after June each, a strong pattern that we wanted to capture within our algorithm.**

## 2.5 Algorithm

We chose to implement XGBoost, a quick and efficient algorithm that has proven to perform really well in data science competitions. Short for “extreme gradient boosting”, It is based on the tree ensemble model, which is a set of classification and regression trees (CART). Our problem is a classification one, as the target variable is a set of 22 binaries, representing whether a product is recommended or not. While a classifier is a method for assigning predictions (the target variable) to some input variables by recursively partitioning the training data space and fitting a simple prediction model within each partition, XGBoost algorithm takes into account a multitude of trees in order to improve the performance.

XGBoost differentiates itself from the other tree ensemble models by the way it is trained. The training happens via the optimization (minimization) of an objective function, made up of two parts: the training loss, which measures how well the model fits on the training data, and the regularisation term, which measures the model complexity and helps to smooth the weights assigned to the trees in order to avoid overfitting.

This optimisation underlines what is known as the bias-variance trade-off. The majority of data science problem solving rely on the assumption that a model shall have an adequate prediction ability, even if it is at the expense of precise predictions on the training data. This avoids overfitting (low bias), the phenomenon which impedes the model from having good generalisation by fitting too closely to the training data.

As the objective function includes functions as parameters, it cannot be optimized with traditional methods. Instead, the model trains additively. Trees that most improve the model (chosen by the exact greedy algorithm) are added to the predictions during many rounds until the regularized objective function (multiclass negative log-likelihood) is minimised.

To let place for other trees and prevent overfitting, two methods are used: the shrinkage parameter (learning rate), which applied to the new added trees to the predictions reduces the individual impact of the trees, and column subsampling, which uses only a subsample ratio of the column for each split.

## 2.6 Parameters

Outlined here are the parameters we specified for our algorithm.

### 2.6.1 Learning task parameters

* **Objective:** ‘multi: softprob’

Specifies the learning task and the corresponding learning objective. We chose ‘multi: softprob’, which stands for multiclass classification using the softProb objective (returning a matrix of classes and probabilities for each customer)

* **Eval\_metric:** ‘map@7’

Chosen validation measure. We choose the same as the one used in the Kaggle problem, the multiclass negative log-likelihood.

* **Seed:** ‘seed\_val’

Value used to generate reproducible results.

### 2.6.2 Booster Parameters

* **Eta:** 0.05

Learning rate. A lower rate gives the model a better generalisation but must be offset computationally by doing more rounds to capture the residuals.

* **Max\_depth:** 6

Maximum depth of a tree. A too high depth might lead to overfitting.

* **Num\_class:** 22

Number of classes to classify by; it’s the number of target variables basically.

* **Min\_child\_weight:** 2

Minimum number of customers to split the tree further.

* **Subsample:** 0.85

Fraction of observations to be randomly sampled for each tree.

* **Colsample\_bytree:** 0.9

Subsample ratio of columns for each split, in each level.

* **Num\_rounds:** 100

Number of rounds.

## 2.7 Performances

The engine in its default configuration returns a score of 0.026 on Kaggle, which is good enough for a top 1000 place finish on the leader boards.

The final model had a score of 0.030, achieving a 170th place in the leader boards, a top 10% finish.

Regarding the parameters tuning, many values have been tested, but there were no clear trends whether augmenting or diminishing the value of a parameter jointly or not would be beneficial to the performance of the algorithm.

## 2.8 Advantages

1. **Regularization**

As the objective function comprises a regularization term working as a penalty term, it prevents the model of overfitting.

1. **Parallel processing**

This makes the algorithm very fast.

1. **High flexibility**

The user can build custom optimization objectives and evaluation criteria.

1. **Handling missing values**

The algorithm can by default impute or skip through null values, minimizing engineering and formatting efforts.

1. **Tree pruning**

The algorithm makes splits up to the Max\_depth specified, starts pruning the tree backwards, and remove splits beyond which there is no positive gain, resulting in the simplest possible method.

## 2.9 Future improvements

* **PCA Analysis**

Perform a PCA analysis to only keep features accounting for most of the variance.

* **K-fold cross-validation**

Implement a k-fold cross-validation, with k=5-10. This is a solution to avoid overfitting on the test set, which can still happen, if the parameters of the model are tweaked until the estimator performs optimally.

In this approach, the training set is split into k smaller sets. A model is then trained using k-1 of the folds as training data. The resulting model is validated on the remaining part of the data (i.e., it is used as a test set to compute a performance measure such as accuracy).

* **Add other features**

Think of other features that might improve the performance. (e.g. Lag\_product correlation measures.)

* **Neural networks**

Experiment a multi-layered neural network model, which might yield improvement regarding performance.